v1.0
20140416

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO
a.a 2013/2014
Lecturer: Alessandro Ricci

[module 2.5] PROCESS COORDINATION: BASIC MECHANISMS

BASIC MECHANISMS & ABSTRACTION FOR PROCESS COORDINATION

- The algorithms for the CS problem described in previous module can be run on a *bare* machine
  - they use only machine language instructions that the computer provides
  - too low level to be used efficiently and reliably

> introduction of basic mechanisms and abstractions for process coordination

  - provided directly by the concurrent machine and used in concurrent languages

- Main constructs
  - **semaphores**
  - **monitors**

SEMAPHORES

- Introduced by Dijkstra in 1968, semaphores are a very simple but powerful general-purpose construct which makes it possible to solve almost *any* mutual exclusion and synchronization problem
  - informally, a semaphore functions as street semaphore, *blocking* and *unblocking* process execution (car movement) according to the need
- Semaphore as a primitive data type provided by the concurrent machine

SEMAPHORE DATA TYPE

- A semaphore S is a compound data type with two fields:
  - **S.V** is an *integer* >= 0
  - **S.L** is a ***set*** of process (id)
- It can be initialized with:
  - a value k >= 0 for S.V
  - the empty set {} for S.L
    - e.g. semaphore `S = (k,{})`
- It provides two basic *atomic* operations
  - **wait(S)**
    - also called **P(S)** from Dijkstra original choice
  - **signal(S)**
    - also called **V(S)** from Dijkstra original choice

WAIT OPERATION

- Definition (p is current process executing wait):

```
wait(S)=
< if (S.V > 0)
    S.V ← S.V - 1
  else
    S.L = S.L + {p}
    p.state ← blocked >
```

- Description
  - if the value of the semaphore V is > 0 (~the semaphore is *green*), then it is simply decremented.
  - otherwise if the value V = 0 (the semaphore is red), then the process is blocked
    - *p is blocked on* the semaphore S
- Note that `wait` is meant to be *atomic*

SIGNAL OPERATION

- Definition:

```
signal(S)=
< if (S.L = {})
    S.V ← S.V + 1
  else
    let q ← arbitrary element of S.L
    S.L ← S.L - {q}
    q.state ← ready >
```

- Description
  - If no process is waiting, then the semaphore value is incremented
  - otherwise select a process q blocked on the semaphore, and unblock it.
- Also `signal` is meant to be *atomic*

SEMAPHORE INVARIANT

- Let k be the initial value of the integer component of the semaphore, #signal(S) the number of signal(S) statements that have been executed, and #wait(S) the number of wait(S) statements that have been completely executed.
  - a process that is blocked when executing wait(S) is not considered to have successfully executed the statement
- **THEOREM**
  - A semaphore S satisfies the following invariants:

```
S.V >= 0
S.V = k + #signal(S) - #wait(S)
```

TYPES OF SEMAPHORES

- **Mutex** or *binary semaphores*
  - semaphores whose integer component can take only two values, 0 and 1
  - the name derives from their typical use for implementing mutual exclusion
- **General** or *counting semaphores*
  - semaphores whose integer component can take any value >= 0
- **Event** semaphores
  - initialized with 0, used for synchronisation purpose

DEFINITIONS OF SEMAPHORES

- There are several different definitions of the semaphore type
  - differences relate to the specification of liveness properties, and do not affect the safety properties that follow from the semaphore invariants
- Main types
  - **strong** vs **weak** semaphores
  - **busy-wait** semaphores

STRONG SEMAPHORES

- In **strong** semaphore S.L is not a set, but a **queue**
  - semaphores in which S.L is a *set* are called *weak*

```
wait(S) =
< if (S.V > 0)
    S.V ← S.V - 1
  else
    append(S.L,p)
    p.state ← blocked >
```

```
signal(S) =
< if (S.L = empty_queue)
    S.V ← S.V + 1
  else
    let q ← take(S.L)
    q.state ← ready >
```

- Important property: **no starvation**
  - for a strong semaphore *starvation is impossible for any number N of processes*

BUSY-WAIT SEMAPHORES

- Semaphores without S.L
  - semaphore operations are still atomic, so there is no interleaving between the two statements implementing the wait(S) operation

```
wait(S) =
< await(S.V > 0)
  S.V ← S.V - 1 >
```

```
signal(S) =
< S.V ← S.V + 1 >
```

- Loosing freedom from starvation
  - with busy-wait semaphores you cannot assume that a process enters in its critical section event in the 2-process solution
- Busy-wait semaphores are appropriate in a multiprocessor system when the waiting process has its own processor and is not wasting CPU time that could be used for other computation
  - they would be appropriate in a system with a little contention so that the waiting process would not waste too much CPU time

SEMAPHORE USAGE

- Semaphores are primitive constructs that can be used as low-level building block to solve almost *any* problem concerning process interaction
  - in shared memory architecture
- In particular they can be used for both:
  - **mutual exclusion**
    - e.g. critical section problem
    - implementing *locks*
    - *...*
  - **synchronization**
    - event semaphore for signaling
    - barriers
    - ...

CRITICAL SECTION WITH SEMAPHORES

- Using a semaphore, the solution of the critical section problem for two processes is trivial
  - using a semaphore as a lock

CS with semaphores: 2 processes	
binary semaphore S ← (**1**,{})	
P	Q
loop forever p1: NCS p2: wait(S) p3: CS p4: signal(S)	loop forever q1: NCS q2: wait(S) q3: CS q4: signal(S)

PROVING CORRECTNESS

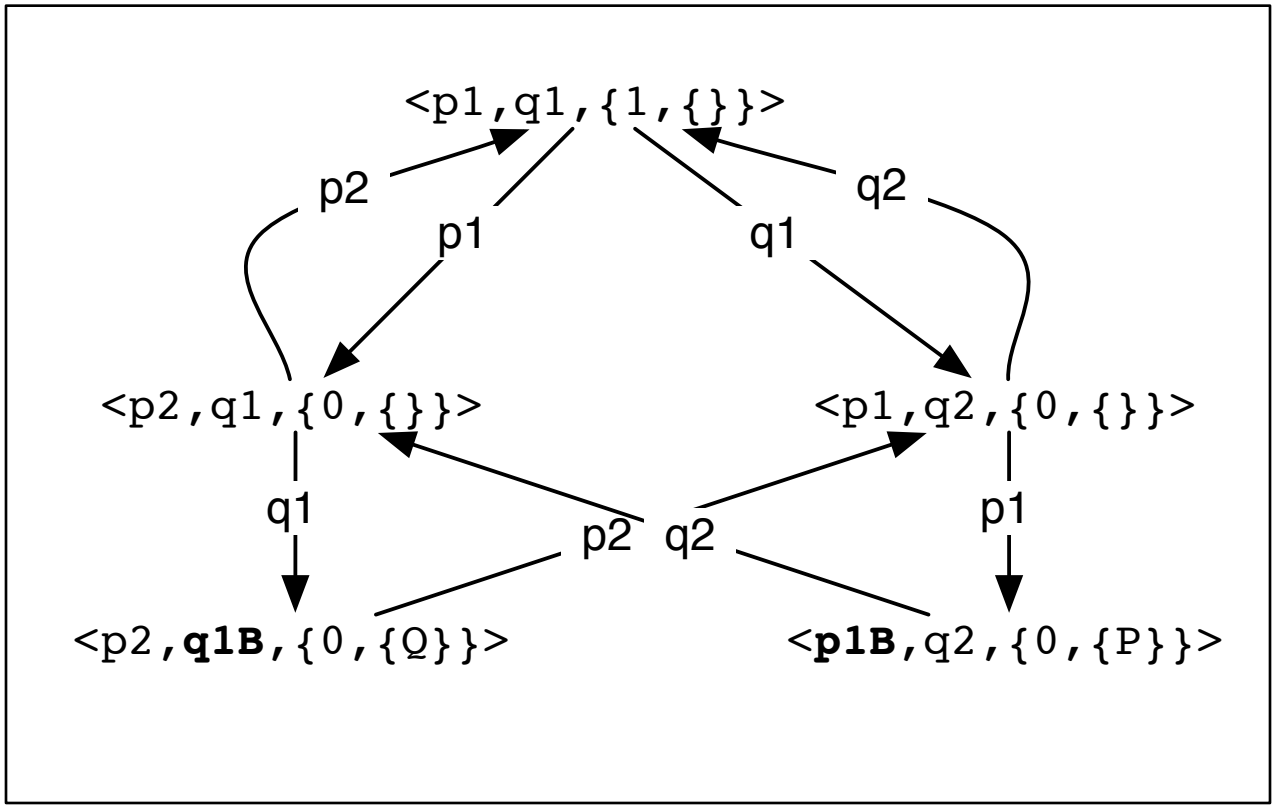
- Building the reduced state diagram and checking properties

CS with semaphores: 2 processes (abbreviated)	
`binary semaphore S ← (1,{})`	
p	**q**

`loop forever`
`p1: wait(S)`
`p2: signal(S)`
	`loop forever`
	`q1: wait(S)`
	`q2: signal(S)`

- It can be verified that the semaphore solution for the CS problem is correct
  - there is mutual exclusion, free from deadlock and starvation

STATE DIAGRAM WITH BLOCKED STATES



- Remarks
  - semaphore structure in the tuple state { s.V, s.L }
  - *blocked state* for P and Q labelled as p1**B** and q1**B**

CRITICAL SECTION FOR N PROCESSES

- The same solution applies also for N processes

CS with semaphores: N processes
`binary semaphore S ← (1,{})`
Any process

`loop forever`
`p1: NCS`
`p2: wait(S)`
`p3: CS`
`p4: signal(S)`

- But it there is no more freedom from starvation

USING SEMAPHORES FOR SYNCHRONIZATION

- Semaphores provide a basic mechanism also to synchronise processes
  - solving order of execution problems

> **event semaphores**

  - used to send / receive a temporal signal
  - initialized to `(0,{})`

- An example: merge sort

Merge sort

```
binary semaphore S1 ← (0,{})
binary semaphore S2 ← (0,{})
integer array A
```

sort1	**sort2**	**merge**

`p1: sort 1st half of A`
`p2: signal(S1)`
	`q1: sort 2nd half of A`
	`q2: signal(S2)`
		`r1: wait(S1)`
		`r2: wait(S2)`
		`r3: merge halves of A`

THE PRODUCER-CONSUMER (P/C) PROBLEM

- The producer-consumer problem is an example of an *order-of-execution problem*
- Two types of processes:
  - **producers**
    - a producer process executes a statement *produce* to create a data element and then sends this element to the consumer process
  - **consumers**
    - upon receipt of a data element from a producer process, a consumer process executes a statement *consume* with the data element as a parameter
- Ubiquitous patterns in CS:

PRODUCER	CONSUMER
Communication line	Web browser
Web browser	Communication line
Keyboard	Operating Systems
Word processor	Printer
Game program	Display screen
...	...

P/C WITH A BUFFER

- When a data element must be sent from one process to another, the communication can be
  - **synchronous**
    - communication cannot take place until both the producer and consumer are ready to do so
  - **asynchronous**
    - the communications channel itself has some capacity for storing data elements
    - uncoupling very useful useful for dynamic / open systems
      - temporal uncoupling among participants
      - dynamic set of processes
    - useful also when producers and consumers have different speed
- The asynchronous case needs the introduction of a proper **buffer** where to store and retrieve data
  - shared data structures with a mutable state, read by consumers and written by producers

P/C + INFINITE BUFFER

- If there is an infinite buffer, there is only one interaction that must be synchronized
  - the consumer must not attempt a take operation from an empty buffer

P/C with infinite buffer

```
UnboundedQueue<Item> buffer ← empty queue
semaphore nAvailItems ← (0,{})
```

producer	**consumer**

`loop forever`
`p1: Item el ← produce`
`p2: append(buffer,el)`
`p3: signal(nAvailItems)`
	`loop forever`
	`q1: wait(nAvailItems)`
	`q2: Item el ← take(buffer)`
	`q3: consume(el)`

- invariant: `nAvailItems.V = #buffer`
  - actually true only if p2+p3 and q1+q2 are considered atomic
- Note that in this example *append* and *take* are meant to be atomic
- `nAvailItems` is called **resource** *semaphore*

P/C + BOUNDED BUFFER

- In this case, there is also another interaction that must be synchronized
  - the producer must not attempt an append operation on a buffer which is full

P/C with *bounded* buffer

```
BoundedQueue<Item> buffer ← empty queue
semaphore nAvailItems ← (0,{})
semaphore nAvailPlaces ← (N,{})
```

producer	**consumer**

`loop forever`
`p1: Item el ← produce`
`p2: wait(nAvailPlaces)`
`p2: append(buffer,el)`
`p3: signal(nAvailItems)`
	`loop forever`
	`q1: wait(nAvailItems)`
	`q2: Item el ← take(buffer)`
	`q3: signal(nAvailPlaces)`
	`q4: consume(el)`

- `nAvailItems` and `nAvailPlaces` are an example of **split** *semaphores*
- invariant: `nAvailItems + nAvailPlaces = N`

COMBINING MUTEX+SYNCH SEMAPHORES

- As a generalization of previous case, we consider the shared use of a *non-atomic* data structure (a buffer in this case), so with non-atomic operations
- introducing a mutex for guaranteeing also mutual exclusion

P/C with *bounded* buffer with multiple producers & consumers

```
BoundedQueue<Item> buffer ← empty queue
semaphore nAvailItems ← (0,{})
semaphore nAvailPlaces ← (N,{})
binary semaphore mutex ← (1,{})
```

producer	**consumer**
loop forever p1: Item el ← produce p2: wait(nAvailPlaces) p3: wait(mutex) p4: append(buffer,el) p5: signal(mutex) p3: signal(nAvailItems)	loop forever q1: wait(nAvailItems) q2: wait(mutex) q3: Item el ← take(buffer) q4: signal(mutex) q4: signal(nAvailPlaces) p4: consume(el)

DINING PHILOSOPHERS

- Classical problem in the field of concurrent programming
  - originated by an examination question set by Dijkstra in 1971 on a synchronization problem where five computers competed for access to five shared tape drive peripherals
    - retold as the dining philosophers problem by Tony Hoare.
  - nowadays it is an entertaining vehicle for comparing various formalism for writing and proving concurrent problems
    - sufficiently simple & challenging
- Description
  - there is a secluded community of five philosophers who engage in only two activities: *thinking* and *eating*
  - meals are taken communally at a table set with 5 plates and 5 forks
  - in the centre of the table there is a bowl of spaghetti that is endlessly replenished.
  - the spaghetti is hopelessly tangled and a philosopher needs *two* forks in order to eat
  - each philosopher may pick up the forks on his left and on his right, *but only one at a time*

DP PROPERTIES

Philosopher

```
loop forever
p1: think
p2: <pre-protocol>
p3: eat
p4: <post-protocol>
```

- The problem is to design pre- and post- protocols to ensure the following properties:
  - a philosopher can eat only if he/she has two forks
  - **mutual exclusion**
    - no two philosophers may hold the same fork simultaneously
  - **freedom from deadlock**
  - **freedom from starvation**
  - efficient behavior in the absence of contention

FIRST ATTEMPT

- Each fork is modeled as a semaphore
  - wait => taking a fork
  - signal => putting down the fork

Dining philosophers (first attempt)

```
semaphore array[0..4] fork ← [1,1,1,1,1]
```

```
loop forever
p1: think
p2: wait(fork[i])
p3: wait(fork[(i+1)%N])
p3: eat
p4: signal(fork[i])
p5: signal(fork[(i+1)%N])
```

- It can be proved that no fork is ever held by two philosophers
- Unfortunately this solution **deadlocks**
  - under an interleaving that has all philosophers pick up their left forks before any of them tries to pick up the right fork

DEADLOCKS

- A situation wherein *two or more competing actions are waiting for the other to finish, and thus neither ever does*
- Coffman *necessary conditions* for a deadlock to occur (1971)
  - **mutual exclusion condition**
    - a resource that cannot be used by more than one process at a time
  - **hold and wait condition**
    - processes already holding resources may request new resources
  - **no preemption condition**
    - no resource can be removed from a process holding it
    - resources can be released only by the explicit action of the process
  - **circular wait condition**
    - two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds
- Deadlock can only occur in systems where all 4 conditions hold true

DEADLOCKS WITH LOCKS

- It happens when
  - multiple threads wait forever due to cyclic locking dependency
  - simplest case
    - when thread A holds lock L and tries to acquire lock M, but at the same time thread B holds M and tries to acquire L, both thread will wait for ever
    - *deadly embrace*

DEADLOCKS DETECTION & RECOVERY

- Deadlocks **detection** and **recovery**
  - adopted in databases
    - databases are designed to detect and recover from deadlocks
    - transactions typically acquire many locks, until they commit
    - not so uncommon for two transactions to deadlock
  - identifying the set of transactions that are deadlocked by analyzing *is-waiting* dependency graph
    - looking for cycles
    - if a cycle is found, a victim is selected and the transaction aborted
- No automated deadlock detection / recovery mechanism in JVM
  - if threads deadlock, *that's all, folks!*
    - we can just shutdown the application
  - "post-mortem" diagnosis support

BACK TO DP - FIRST SOLUTION: TICKETS

- To ensure liveness we can limit the number of philosophers eating simultaneously (or entering the dining room)
  - introducing *meal (or room) tickets*
  - N-1 tickets for N philosophers

Dining philosophers (solution)

```
semaphore array[0..4] fork ← [1,1,1,1,1]
semaphore ticket ← (4,{})
```

```
loop forever
p1: think
p2: wait(ticket)
p3: wait(fork[i])
p4: wait(fork[(i+1)%N])
p5: eat
p6: signal(fork[i])
p7: signal(fork[(i+1)%N])
p8: signal(ticket)
```

- It can be proved that this solution satisfies all the properties

2ND SOLUTION: BREAKING THE WAIT-FOR CHAIN

- It can be observed that there is no more deadlock if the last philosopher picks up first the right fork and then left one
  - breaking the wait-for chain
- Remark
  - actually - given the total order among the identifiers of the forks - the last philosopher was picking the forks in the opposite order with respect to the other philosophers
    - first the (N-1) fork and then the 0 fork
  - the solution is then about picking the forks always in the same order

2ND SOLUTION: CODE

Dining philosophers (2nd solution)

```
semaphore array[0..4] fork ← [1,1,1,1,1]
```

```
integer first = min(i,(i+1)%N)
integer second = max(i,(i+1)%N)

loop forever
p1: think
p2: wait(fork[first])
p3: wait(fork[second])
p4: eat
p5: signal(fork[first])
p6: signal(fork[second])
```

GENERAL RULE FOR DEADLOCK

- General setting
  - N processes sharing and acquiring multiple locks
- General rules to avoid deadlock
  1. assign a total order to locks
  2. acquire the locks *always in the same order*
- It works because it makes it impossible to have circular wait-for dependency among processes
  - necessary condition for deadlock

READERS-AND-WRITERS PROBLEM

- The problem of readers-writers is similar to the mutual exclusion problem in that several processes are competing for access to a critical section [Courtois, Heymans, Parnas - 1971].
- In this problem, however, we divide the processes into two classes:
  - **Readers**
    - which are required to exclude writers but not other readers
  - **Writers**
    - which are required to exclude both readers and other writers
- The problem is an abstraction of *access to databases* (or any kind of shared resource)
  - no danger in having process reading data concurrently
  - writing or modifying data must be done under mutual exclusion to ensure consistency of the data
- Solutions must satisfy these **invariants**

```
nR >= 0
nW = 0 || nW = 1
(nR > 0 → nW = 0) ∧ ( nW = 1 → nR = 0)
```

nR = number of readers, nW = number of writers

AN OVER-CONSTRAINED SOLUTION

- Using a single semaphore functioning as a *lock*

Readers-and-writers: first attempt

```
binary semaphore rw ← (1,{})
DataBase dbase;
```

reader	**writer**
loop forever p1: wait(rw) p2: Item el ← read(dbase) p3: signal(rw)	loop forever q1: wait(rw) q2: Item el ← create_record; q3: write(dbase,el) q4: signal(rw)

- Each reader and writer has exclusive access to the dbase
  - **over-constrained** solution: serializing access also for readers!

SOLUTION

- Readers don't use the same lock of writers
  - `mutexR` lock for readers for updating common data structures (`nr`)

Readers-and-writers: solution

```
binary semaphore mutexR ← (1,{})
int nr ← 0
binary semaphore rw ← (1,{})
DataBase dbase;
```

reader

```
loop forever
p1:  wait(mutexR)
p2:  if (nr == 0)
p3:    wait(rw)
p4:  nr ← nr + 1
p5:  signal(mutexR)
p6:  Item el ← read(dbase)
p7:  wait(mutexR)
p8:  nr ← nr - 1
p9:  if (nr == 0)
p10:   signal(rw)
p11: signal(mutexR)
```

writer

```
loop forever
q1: wait(rw)
q2: Item el ← create_record;
q3: write(dbase,el)
q4: signal(rw)
```

THE CIGARETTE SMOKER'S PROBLEM

- Synchronization problem proposed by S. Patil in 1971, to investigate the limits of the semaphore primitive
- Problem statement
  - assume that there is a group of four people: 3 *smokers* and 1 *agent (arbiter).* To roll and smoke a cigarette three ingredients are needed: paper, tobacco, matches. One of the smokers has an infinite supply of papers, another has an infinite supply of tobacco, and another has an infinite supply of matches. The agent has an infinite supply of all three ingredients.
  - the four participants repeatedly perform the following: the agent puts two ingredients on the table; the smoker who has the remaining ingredient takes the two ingredients, rolls a cigarette, smokes it, and notifies the agent on completion. Then the agent puts another two ingredients on the table, and so on
  - the problem is to write a program to synchronize the agent and the smokers

PATIL'S ARGUMENT

- Patil's argument was that Edsger Dijkstra's semaphore primitives were limited
  - he used the cigarette smokers problem to illustrate this point by saying that it cannot be solved with semaphores.
- However, Patil placed heavy constraints on his argument:
  - the process code is the following (and is not modifiable)

```
shared S: array[1..3] of binary semaphores, initially all 0
       agent: binary semaphore, initially 1
local i,j: range over [1,2,3]
loop
  set i and j (at random) to two different values from [1,2,3]
  wait(agent)
  signal(S[i])
  signal(S[j])
end_loop
```

  - the solution is not allowed to use conditional statements or an array of semaphores.
- With these two constraints, a semaphore-based solution to the cigarette smokers problem is impossible.

BEYOND SEMAPHORES...

- Semaphores are a powerful construct, but very low level
  - error-prone programs
  - hard to use in complex concurrent programs

> looking for high-level constructs: ***monitors***
  - introduces by Brinch Hansen (1973)
  - generalized by Hoare (1974)

MONITORS

- def. **Monitor**
  - a concurrent programming data abstraction encapsulating the *synchronization and mutual exclusion policy in accessing it*
    - state + operations + concurrency policy
    - like a *module* + basic mechanisms to enforce correctness in module concurrent access
- Generalization of the *kernel* or *supervisor* concept in operating systems, where critical sections such as the allocation of memory are centralized in a privileged program
  - applications programs request services which are performed by the kernel
  - kernels are run in a HW mode that ensures that they cannot be interfered with by application programs
  - monitors as decentralized versions of the monolithic kernel
- Generalization of the *object* notion in OOP
  - classes encapsulating data + operation + synchronization / mutex policy

MONITOR DEFINITION

- Monitor are declared and created in different ways according to the specific language.

- An abstract representation:

```
monitor MonitorName {

  declaration of permanent variables

  initialization statements

  procedures (or entries)

}
```

MONITOR PROPERTIES

- Monitors as instances of abstract data type
  - *only operations (procedures) name are visible outside the monitor*
    - they are the *interface*
    - they provide the only gates through the “wall” defined by the monitor declaration
    - call to monitor procedures:
      `call MonitorName.OpName(params)`
      (often written simply
      `MonitorName.OpName(params))`
  - statements within the monitor cannot access variables declared *outside* del monitor
  - permanent variables are initialized before any procedure is called

MONITOR FEATURES: MUTUAL EXCLUSION

- **Intrinsic / implicit mutual exclusion**
  - procedures *by definition* execute with mutual exclusion
    - a monitor procedure is called by an external process
    - a procedure is active if some process is executing a statement in the procedure
    - *at most one instance of one monitor procedure may be active at a time*
    - processes that find the monitor ‘busy’ are suspended

SIMPLE EXAMPLE

- Classic counter
  - but thread-safe, thanks to monitor properties

```
monitor Counter {

  int count;

  procedure inc(){
    count := count + 1
  }

  procedure getValue():int {
    return count;
  }
}
```

REMARKS

- The mutual exclusion is **implicit** and does not require the programmers to use any other mechanism (such as wait and signal..)
  - if operations of the same monitor are called by more than one process, the implementation ensures that these are executed under mutual exclusion
    - operations are executed **atomically** (with respect to each other)
  - if operations of different monitors are called, their execution can be interleaved
- There is no explicit queue associated with the monitor procedure
  - *starvation* problem

MONITOR FEATURES: SYNCHRONIZATION

- **Explicit synchronization support**
  - through ***condition variables***
    - used inside the monitors by the programmers to delay a process that cannot safely continue executing until the monitor's state satisfies some boolean condition
    - used also to awake a delayed process when the condition becomes true

CONDITION VARIABLES

- Primitive data types that can be used to suspend (wait) and resume (signal) processes inside a monitor
  - representing *conditions* (events) on the monitor state that wait to be satisfied and that becomes satisfied
- Two basic atomic operations, **waitC** and **signalC**
  - sometimes written simply wait and signal
- Each condition variable is associated with a FIFO queue of blocked processes

COND. VARIABLE OPERATIONS

- **waitC**(cond)
  - suspend the execution of the process and release lock of the monitor
  - abstract implementation:

```
waitC(cond) =
< append p to cond.queue
  p.state := blocked
  monitor.lock := release >
```

- **signalC**(cond)
  - unblock a process waiting on a condition
  - abstract implementation:

```
signalC(cond) =
< if cond.queue != empty
    q := remove head of cond.queue
    q.state := ready >
```

SIMPLE SYNCH. EXAMPLE

- Synchronized cell

```
monitor SynchCell {

  int value;
  boolean available := false;
  cond isAvail;

  procedure set(int v){
    value := v
    available := true
    signalC(isAvail)
  }

  procedure get():int {
    if (!available)
      waitC(isAvail)
    return value
  }
}
```

REMARK

- There is an explicit link between condition variables and their encapsulating monitor

wait operation releases the monitor lock

- This is essential to avoid that a process executing a waitC would block the access to the monitor

OTHER PRIMITIVES

- **emptyC**(cond)
  - check if the queue is empty
- **signalAllC**(cond)
  - like signal, but all the processes waiting on the condition are resumed
- **waitC**(cond,rank)
  - wait in order of increasing value of rank
- **minrank**(cond)
  - returns the value of rank of process at front of wait queue

SYNCH CELL REVISITED /2

- Using signalAllC to wake up every process in the queue

```
monitor ImprovedSynchCell {

  int value;
  boolean available;
  cond isAvail;

  procedure set(int v){
    value := v
    available := true
    signalAllC(isAvail)
  }

  procedure get():int {
    if (!available)
      waitC(isAvail)
    return value
  }
}
```

SYNCH CELL REVISITED /3

- The same behavior can be obtained without signalAllC
  - signaling as soon as a suspended process is awaken

```
monitor ImprovedSynchCell {

  int value;
  boolean available;
  cond isAvail;

  procedure set(int v){
    value := v
    available := true
    signalC(isAvail)
  }

  procedure get():int {
    if (!available){
      waitC(isAvail)
      signalC(isAvail)
    }
    return value
  }
}
```

IMPLEMENTING A SEMAPHORE

- Two implementations of a semaphore using monitors

```
monitor Semaphore {
  integer s := <InitValue>
  cond notZero

  procedure wait(){
    if s = 0
      waitC(notZero)
    s := s - 1
  }
  procedure signal(){
    s := s + 1
    signalC(notZero)
  }
}
```

```
monitor Semaphore {
  integer s := <InitValue>
  cond notZero

  procedure wait(){
    if s = 0
      waitC(notZero)
    else
      s := s - 1
  }
  procedure signal(){
    if emptyC(notZero)
      s := s + 1
    else
      signalC(notZero)
  }
}
```

SEMAPHORES VS. CONDITION VARIABLE IN MONITORS

SEMAPHORE	MONITOR
wait may or may not block	waitC always blocks
signal always has an effect	signalC has no effect if queue is empty
signal unblocks an arbitrary blocked process	signalC unblocks the process at the head of the queue
a process unblocked by signal can resume execution immediately	depending on the specific signaling semantics, a process unblocked by signalC must wait for the signaling process to leave the monitor

SIGNALING DISCIPLINE

- When a process executes a signal, even if there could be multiple processes ready to execute within the monitor, *only one process can have exclusive access*
  - because of the basic semantics of monitors
  - only one process is chosen to keep active
  - > either the signaling or the waiting process can be resumed, not both
    - classic solution for monitors

SIGNALING DISCIPLINE: SEMANTICS

- **Signal and Continue**
  - the signaler continues and the signaled process executes at some later time
  - nonpreemptive
- **Signal and Wait**
  - signaled process executes now and the signaler waits, eventually competing with other processes waiting for entering the monitor
  - preemptive
- **Signal and Urgent Wait** *(or Immediate Resumption Requirement)*
  - like signal and wait, but the signaler has priority over processes waiting for the lock
  - classic solution for monitors

SIGNALING DISCIPLINE: SEMANTICS

- Given
  - **S** = precedence of the signaling processes
  - **W** = precedence of the waiting processes
  - **E** = precedence of processes blocked on an procedure

- Signal and Continue
  - **E < W < S**

```
while (!B)
  wait(cond)
<access>
```

- Signal and Wait
  - **E = S < W**

```
if (!B)
  wait(cond)
<access>
```

- Signal and Urgent Wait
  - **E < S < W**

USING MONITORS

- Monitors can be used to implement any resource or data structure which is used concurrently by multiple processes and in which we want to encapsulate the synchronization policies
- Revisiting the main examples
  - *Producers-Consumers*
    - implementing the bounded-buffer as a monitor
  - *Readers-and-Writers*
    - implementing the rw-lock as a monitor
  - *Resource allocation and management*
    - implementing the resource allocator as a monitor

PRODUCERS-CONSUMERS

```
monitor BoundedBuffer {

  ElemType buffer := <EmptyBuffer>
  cond notFull, notEmpty;

  procedure put(ElemType elem){
    if (buffer is full)
      waitC(notFull)
    append(buffer,elem)
    signalC(notEmpty)
  }

  procedure take(): ElemType {
    if (buffer is empty)
      waitC(notEmpty)
    ElemType el := head(buffer)
   signalC(notFull)
   return el
  }
}
```

Producer	Consumer
`loop`	`loop`
`p1: ElemType el := produce`	`q1: ElemType el := BoundedBuffer.take()`
`p2: BoundedBuffer.put(el)`	`q2: consume(el)`

PRODUCERS-CONSUMERS

- Using a circular array for implementing the buffer data structure ...

```
monitor BoundedBuffer {

  int[] elems := new int[MAX_ELEMS]
  int first := 0, last := 0
  cond notFull, notEmpty

  procedure put(int elem){
    if ((last + 1) % MAX_ELEMS) = first
      waitC(notFull)
    elems[last] = elem
    last := (last + 1) % MAX_ELEMS
    signalC(notEmpty)
  }

  procedure take(): int {
    if (first = last)
      waitC(notEmpty)
    int elem = elems[first]
    first = (first + 1) % MAX_ELEMS
    signalC(notFull)
    return elem
  }
}
```

READERS-AND-WRITERS (signal-and-continue)

```
monitor RWLock {
  int nr, nw = 0;
  cond okToRead,okToWrite;

  procedure void request_read(){
    while (nw > 0)
      waitC(okToRead);
    nr := nr + 1;
  }
  procedure void  release_read(){
    nr := nr - 1;
    if (nr = 0)
      signalC(okToWrite)
  }
  procedure void request_write(){
    while (nr > 0 or nw > 0)
      waitC(okToWrite)
    nw := nw + 1;
  }
  procedure void release_write(){
    nw := nw - 1;
    signalC(okToWrite);
    signalAllC(okToRead);
  }
}
```

Invariant:

```
(nr == 0 or nw == 0) and (nw <= 1)
```

READERS-AND-WRITERS solution #2

```
monitor RWLock {
 integer readers := 0
 integer writers := 0
 cond okToRead,okToWrite;

 procedure startRead(){
   if writers != 0
     waitC(okToRead)
   readers := readers + 1
   signalC(okToRead)
 }
 procedure endRead(){
   readers := readers - 1
   if readers = 0
     signalC(okToWrite)
 }
 procedure startWrite(){
   if writers != 0 or readers != 0
     waitC(okToWrite)
   writers := writers + 1
 }
 procedure endWrite(){
   writers := writers - 1
   if emptyC(okToRead)
     then signalC(okToWrite)
     else signalC(okToRead)
 }
}
```

Reader	Writer
`p1: RWLock.startRead`	`q1: RWLock.startWrite`
`p2: read the dbase`	`q2: write the dbase`
`p3: RWLock.endRead`	`q3: RWLock.endWrite`

RESOURCE ALLOCATION

- Monitors are typically used to function as **resource allocator**
  - enforcing some policy in allocating resources to processes requesting resource access

```
monitor ResourceAllocator {

  procedure request(<Params>){
    < block the request
      until the resource or a resource
      is available, according to some policy >
  }

  procedure release(<Params>){
    < possibly unblock some pending request >
  }
}
```

EXAMPLE: SHORTEST-JOB-NEXT SCHEDULING

- Allocator applying the *Shortest-Job-First*:

```
monitor SJFAllocator {
  bool free = true;
  cond turn;

  procedure request(int time){
    if (free)
      free = false;
    else
      waitC(turn,time);
  }

  procedure release(){
    if (emptyC(turn))
      free = true;
    else
      signalC(turn)
  }
}
```

Invariant:

```
turn ordered by time AND
(free => turn is empty)
```

MONITOR IMPLEMENTATION

- Monitor can be realized using semaphores, in particular
  - one semaphore `mutex` for mutual exclusion
  - for each condition variable, a semaphore `condsem` and a counter `condcount` keeping track of the number of processes suspended on the variable

Signal and Continue semantics:

Prologue for each operation:
```
wait(mutex)
```
Epilogue for each operation:
```
signal(mutex)
```

```
waitC(cond) =
  condcount++;
  signal(mutex);
  wait(condsem);
  wait(mutex);

signalC(cond) =
  if (condcount > 0){
    condcount--;
    signal(condsem)
  }
```

Signal and Wait semantics:

Prologue for each operation:
```
wait(mutex)
```
Epilogue for each operation:
```
signal(mutex)
```

```
waitC(cond) =
  condcount++;
  signal(mutex);
  wait(condsem);

signalC(cond) =
  if (condcount > 0){
    condcount--;
    signal(condsem);
    wait(mutex);
  }
```

BUILDING REUSABLE COORDINATION COMPONENTS AS MONITORS

- Exploiting monitors to realize reusable synchronization / coordination components
  - latches
  - barriers
  - rendez-vous
  - message boxes
  - blackboards
  - event services
  - ...